CSE 591/392: GPU Programming

Basics on Architecture and Programming

Klaus Mueller

Computer Science Department
Stony Brook University
Recommended Literature

Text book

Reference books

Programming Massively Parallel Processors
A Hands-on Approach

CUDA by Example
An Introduction to General-Purpose GPU Programming

CUDA Programming
A Developer's Guide to Parallel Computing with GPUs

The Art of Multiprocessor Programming

Patterns for Parallel Programming

NVIDIA CUDA
Programming Guide

NVIDIA CUDA™
Programming Guide

Processing Guide

Best Practices Guide

version 2.3.1

Programming guides available from nvidia.com

more general books on parallel programming
Course Topic Tag Cloud

- Architecture
- Limits of parallel programming
- Host
- Performance tuning
- Debugging
- Thread management
- Algorithms
- Memory
- CUDA
- Device control
- OpenCL
- Parallel programming
- Example applications
Speedup Curves

GPU Performance Trends

- NVIDIA GPU
- Intel CPU

Trends:
- 5800
- 6800 Ultra
- 7800 GTX
- 8800 Ultra
- 8800 GTX
- 7900 GTX
- GTX 280
- GTX 285
- Intel Xeon Quad-core 3 GHz

Timeline:
- 9/22/02
- 2/4/04
- 6/18/05
- 10/31/06
- 3/14/08

GFlop/s
but wait, there is more to this.....
Amdahl’s Law

Governs theoretical speedup

\[
S = \frac{1}{(1 - P) + \frac{P}{S_{parallel}}} = \frac{1}{(1 - P) + \frac{P}{N}}
\]

P: parallelizable portion of the program
S: speedup
N: number of parallel processors
Amdahl’s Law

Governs theoretical speedup

\[ S = \frac{1}{(1 - P) + \frac{P}{S_{parallel}}} = \frac{1}{(1 - P) + \frac{P}{N}} \]

P: parallelizable portion of the program
S: speedup
N: number of parallel processors

P determines theoretically achievable speedup

- example (assuming infinite N): P=90% → S=10
  P=99% → S=100
Amdahl’s Law

How many processors to use

- when $P$ is small $\rightarrow$ a small number of processors will do
- when $P$ is large (embarrassingly parallel) $\rightarrow$ high $N$ is useful
Focus Efforts on Most Beneficial

Optimize program portion with most ‘bang for the buck’

• look at each program component
• don’t be ambitious in the wrong place
Focus Efforts on Most Beneficial

Optimize program portion with most ‘bang for the buck’
• look at each program component
• don’t be ambitious in the wrong place

Example:
• program with 2 independent parts: A, B (execution time shown)

```
  A                  B
Original program

  B sped up 5×

  A sped up 2×
```

• sometimes one gains more with less
Limits from mismatch of parallel program and parallel platform

- man-made ‘laws’ subject to change with new architectures
Limits from mismatch of parallel program and parallel platform
  • man-made ‘laws’ subject to change with new architectures

Memory access patterns
  • data access locality and strides vs. memory banks
Beyond Theory....

Limits from mismatch of parallel program and parallel platform
  • man-made ‘laws’ subject to change with new architectures

Memory access patterns
  • data access locality and strides vs. memory banks

Memory access efficiency
  • arithmetic intensity vs. cache sizes and hierarchies
Beyond Theory....

Limits from mismatch of parallel program and parallel platform
  • man-made ‘laws’ subject to change with new architectures

Memory access patterns
  • data access locality and strides vs. memory banks

Memory access efficiency
  • arithmetic intensity vs. cache sizes and hierarchies

Enabled granularity of program parallelism
  • MIMD vs. SIMD
Limits from mismatch of parallel program and parallel platform
  • man-made ‘laws’ subject to change with new architectures

Memory access patterns
  • data access locality and strides vs. memory banks

Memory access efficiency
  • arithmetic intensity vs. cache sizes and hierarchies

Enabled granularity of program parallelism
  • MIMD vs. SIMD

Hardware support for specific tasks → on-chip ASICS
Limits from mismatch of parallel program and parallel platform
  • man-made ‘laws’ subject to change with new architectures

Memory access patterns
  • data access locality and strides vs. memory banks

Memory access efficiency
  • arithmetic intensity vs. cache sizes and hierarchies

Enabled granularity of program parallelism
  • MIMD vs. SIMD

Hardware support for specific tasks → on-chip ASICS
Support for hardware access → drivers, APIs
Transferring the data to the device is also important

- computational benefit of a transfer plays a large role
- transfer costs are (or can be) significant
Device Transfer Costs

Transferring the data to the device is also important

• computational benefit of a transfer plays a large role
• transfer costs are (or can be ) significant

Adding two \((N \times N)\) matrices:

• transfer back and from device: 3 \(N^2\) elements
• number of additions: \(N^2\)

\[ \text{operations-transfer ratio} = 1/3 \text{ or } O(1) \]
Device Transfer Costs

Transferring the data to the device is also important
• computational benefit of a transfer plays a large role
• transfer costs are (or can be) significant

Adding two \((N \times N)\) matrices:
• transfer back and from device: \(3 \, N^2\) elements
• number of additions: \(N^2\)
→ operations-transfer ratio = 1/3 or \(O(1)\)

Multiplying two \((N \times N)\) matrices:
• transfer back and from device: \(3 \, N^2\) elements
• number of multiplications and additions: \(N^3\)
→ operations-transfer ratio = \(O(N)\) grows with \(N\)
Use GPU to complement CPU execution

- recognize parallel program segments and only parallelize these
- leave the sequential (serial) portions on the CPU

PPP (Peach of Parallel Programming – Kirk/Hwu)
Types of Parallelism

Task based parallelism

- unrelated processes are executed in parallel
- slowest process determines the speed
- also known as *coarse grained parallelism*
- MIMD model = Multiple Instructions Multiple Data

Data based parallelism

- decompose a specific task into *threads*
- each thread executes the same statement at the same time
- also known as *fine grained parallelism*
- SIMD model = Single Instructions Multiple Data
Patterns of Parallelism

Loops
• *for* and *while* statements

Fork and Join

Tiling and grids
• break the domain into sub-problems that map well to the hardware
• 2D tiles/grid for images, 3D tiles/grid for volumes

Divide and Conquer
• recursion: can present problems for parallelism when too deep
• better use an iterative approach that solves a level in parallel
Locality

Temporal locality
• data that was accessed before will be likely accessed again
• use cache to reduce access latencies

Spatial locality
• data close to the data accessed last will likely be accessed soon
• fetch entire cache lines when accessing one element

Dirty cache
• a cache location that was written by a process
• update may conflict with the cache of a different process
• need to write back to a shared level of the cache hierarchy

Cache hierarchies
• each level slower then the one below
• scope (to parallel processes) increases with increasing levels
• so must pick the level with sufficient scope
Architectures

von Neumann architecture of traditional CPUs
  • serial instruction decode

Connection machine
  • pioneered by Thinking Machines
  • 4-connected processors and communication

IBM Cell processor
  • PowerPC processors
Multi-Node Computing

Connect several PCs (nodes)

- wiring by fast Ethernet, InfiniBand
- program using OpenMP or MPI (Message Passing Interface)
Course Topic Tag Cloud

Architecture

Limits of parallel programming

Host

Performance tuning

Kernels

Debugging

Thread management

OpenCL

Algorithms

Memory

CUDA

Device control

Example applications

Parallel programming
Overall GPU Architecture (G80)

Memory bandwidth: 86.4 GB/s (GPU)
4GB/s BW (GPU ↔ CPU, PCI Express)

- Host
- Input Assembler
- Thread Execution Manager
- Parallel Data Cache
- Texture
- Load/store
- Global Memory

- SM block
- Streaming multi-processor SM
- Stream processor SP

768 MB Off-chip (GDDR) DRAM (on-board)
GPU Architecture Specifics

Additional hardware
• each SP has a multiply-add (MAD) and one extra multiply unit
• special floating-point function units (SQRT, TRIG, ..)

Massive multi-thread support
• CPUs typically run 2 or 4 threads/core
• G80 can run up to 768 threads/SM → 12,000 threads/chip
• GT200 can run 1024 threads/SM → 30,000 threads/ship

G80 (2008)
• GeForce 8-series (8800 GTX, etc)
• 128 SP (16 SM × 8 SM)
• 500 Gflops (768 MB DRAM)

GT200 (2009)
• GeForce GTX 280, etc
• 240 SP
• 1 Tflops (1 GB DRAM)

NVIDIA Quadro: professional version of consumer GeForce series
NVIDIA Fermi Architecture

GeForce 400 series

- GTX 480, etc
- up to 512 SP (16 × 32) but typically < 500 (GTX 480 has 496 SP)
- 1.3 Tflops (1.5GB DRAM)

Important features:

- C++, support for C, Fortran, Java, Python, OpenCL, DirectCompute
- ECC (Error Correcting Code) memory (Tesla only)
- 512 CUDA Cores™ with new IEEE 754-2008 floating-point standard
- 8× peak double precision arithmetic performance over last-gen GPUs
- NVIDIA Parallel DataCache™ cache hierarchy
- NVIDIA GigaThread™ for concurrent kernel execution
NVIDIA Fermi

On chip:
- SMs: 16
- CUDA cores: $2 \times 16$/SM, 512/chip
- Memory interfaces: 6 (BW 384 bits)
NVIDIA Fermi

- Two 16-wide cores
- Full cross-bar interface
- 4 special function units (math, etc.)
Course Topic Tag Cloud

Architecture

Limits of parallel programming

Host

Performance tuning

Kernels

Debugging

Thread management

OpenCL

Algorithms

Memory

CUDA

Device control

Example applications

Parallel programming
Parallelism is exposed as *threads*

- all threads run the same code
- a thread runs on one SP
- SPMD (Single Process, Multiple Data)

The threads divide into blocks

- threads execute together in a block
- each block has a unique ID within a grid → *block ID*
- each thread has a unique ID within a block → *thread ID*
- block ID and thread ID can be used to compute a *global ID*

The blocks aggregate into *grid cells*

- each such cell is a SM
Thread communication

- threads within a block cooperate via shared memory, atomic operations, barrier synchronization
- threads in different blocks cannot cooperate
Threads within a block are organized into SPMD **warps**

- execute the same instruction simultaneously with different data

**A warp is 32 threads**

→ a 16-core block takes 2 clock cycles to compute a warp

**One SM can maintain 48 warps simultaneously**

- keep one warp active while 47 wait for memory → latency hiding
- 32 threads × 48 warps × 16 SMs → 24,576 threads!
Mapping

- depends on device hardware

Thread management

- very lightweight thread creation, scheduling
- in contrast, on the CPU thread management is very heavy
Function qualifiers

- specify whether a function executes on the host or on the device
- __global__ defines a kernel function (must return void)
- __device__ and __host__ can be used together

<table>
<thead>
<tr>
<th>Function</th>
<th>Exe on</th>
<th>Call from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>GPU</td>
<td>GPU</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>CPU</td>
<td>CPU</td>
</tr>
</tbody>
</table>

CPU=host and GPU=device

For function executed on the device

- no recursion
- no static variable declarations inside the function
- no variable number of arguments
Variable qualifiers

- specify the memory location on the device of a variable
- `__shared__` and `__constant__` are optionally used together with `__device__`

<table>
<thead>
<tr>
<th>Variable</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shared</strong></td>
<td>Shared</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>
Define function as device kernel to be called from the host:

```c
__global__ void KernelFunc(...);
```

Configuring thread layout and memory:

```c
dim3 DimGrid(100,50);  // 5000 thread blocks
dim3 DimBlock(4,8,8);  // 256 threads per (3D) block
size_t SharedMemBytes = 64;  // 64 bytes of shared memory
```

Launch the kernel (<<, >> are CUDA runtime directives)

```c
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```
Program Constructs

Synchronization

• any call to a kernel function is asynchronous from CUDA 1.0 on
• explicit synchronization needed for blocking

Memory allocation on the device

• use cudaMalloc(*mem, size)
• resulting pointer may be manipulated on the host but allocated memory cannot be accessed from the host

Data transfer from and to the device

• use cudaMemcpy(devicePtr, hostPtr, size, HtoD) for host→device
• use cudaMemcpy(hostPtr, devicePtr, size,DtoH) for device→host

Number of CUDA devices

• use cudaGetDeviceCount(&count);
Get device properties for \( \text{cnt} \) devices

for (i=0; i<\text{cnt}, i++)
    cudaGetDeviceProperties(&prop,i);

Some useful device properties (see CUDA spec for more)

- totalGlobalMemory
- warpSize
- maxGridSize
- multiProcessorCount
- ....
```c
void vectorAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i]; }

int main() {
    int N = 4096;

    // allocate and initialize memory
    float *A = (float *) malloc(sizeof(float)*N);
    float *B = (float *) malloc(sizeof(float)*N);
    float *C = (float *) malloc(sizeof(float)*N);
    init(A); init(B);

    vectorAdd(A, B, C, N); // run kernel
    free(A); free(B); free(C); // free memory
```
Example: Vector Add (GPU)

```c
__global__ void gpuVecAdd(float *A, float *B, float *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
}
```

```plaintext
tid = blockIdx.x * blockDim.x + threadIdx.x
```
int main() {
    int N = 4096;  // allocate and initialize memory on the CPU
    float *A = (float *) malloc(sizeof(float)*N); *B = (float *) malloc(sizeof(float)*N); *C = (float*)malloc(sizeof(float)*N)
    init(A); init(B);
    // allocate and initialize memory on the GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, sizeof(float)*N); cudaMalloc(&d_B, sizeof(float)*N); cudaMalloc(&d_C, sizeof(float)*N);
    cudaMemcpy(d_A, A, sizeof(float)*N, HtoD); cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
    // configure threads
    dim3 dimBlock(32,1);
    dim3 dimGrid(N/32,1);
    // run kernel on GPU
    gpuVecAdd <<< dimBlock,dimGrid >>> (d_A, d_B, d_C);
    // copy result back to CPU
    cudaMemcpy(C, d_C, sizeof(float)*N,DtoH);
    // free memory on CPU and GPU
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(A); free(B); free(C); }

Large Sums

Add up a large set of numbers

- Normalization factor:

\[ S = v_1 + v_2 + \cdots + v_n \]

- Mean square error:

\[ MSE = \frac{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2}{n} \]

- L2 Norm:

\[ \| \mathbf{x} \| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \]
Large Sums

Common operator:

\[ \sum: v_1 + v_2 + v_3 + \cdots + v_n \]

Code in C++ running on CPU:

\[ O(n) \] additions

```c++
float sum = 0;
for (int i=0; i<n; i++)
{
    sum += v[i];
}
return sum;
```
Non-Parallel Approach

Non-parallel approach:

Input numbers:

\[
\begin{array}{cccccccccccccc}
10 & 1 & 8 & -1 & 0 & -2 & 3 & 5 & -2 & -3 & 2 & 7 & 0 & 11 & 0 & 2 \\
\end{array}
\]

Generate only one thread

Memory space

\[O(n)\] complexity

How to optimize?
Parallel Approach

Two tasks:

- read numbers to memory
- do the computation (addition) and write result

\[
\begin{array}{cccccccccccccc}
10 & 1 & 8 & -1 & 0 & -2 & 3 & 5 & -2 & -3 & 2 & 7 & 0 & 11 & 0 & 2 \\
\end{array}
\]

Reduction approach
Reduction Approach: Kernel 1

Parallel Approach: Kernel 1

Threads in same step execute in parallel

$O(\log n)$ complexity
Reduction Approach: Kernel 1

CUDA code:

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

very inefficient statement, % operator is very slow
Kernel optimization

Kernel optimization

Kernel optimization

Kernel optimization
Toward Final Optimized Kernel

Performance for 4M numbers:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
</tbody>
</table>

Final optimized kernel:

Kernel 7: multiple elements per thread

Parallel Reduction
Hardware Requirements

NVIDIA CUDA-able devices:
• desktop machines: GeForce 8-series and up
• mobile: GeForce 8m-series and up
• for more information see http://en.wikipedia.org/wiki/CUDA

May use CUDA emulator for older devices
• slower but better debugging support